Hardware Support for Synchronization in the Scalable Coherent Interface (SCI)

Nagi Aboulenein
James Goodman
Stein Gjessing
Philip J. Woest

Technical Report #1117

November 1992

Supersedes TR # 994

Hardware Support for Synchronization in the Scalable Coherent Interface (SCI)

Nagi M. Aboulenein
Computer Sciences Department
University of Wisconsin–Madison
Madison, Wisconsin 53706

Stein Gjessing
Department of Informantics
University of Oslo
Oslo, Norway

James R. Goodman
Computer Sciences Department
University of Wisconsin–Madison
Madison, Wisconsin 53706

Philip J. Woest
Department of EECS
Northwestern University
Evanston, Illinois 60208-3118

November 3, 1992

Abstract

The exploitation of the inherent parallelism in applications depends critically on the efficiency of the synchronization and data exchange primitives provided by the hardware. This paper discusses and analyses such primitives as they are implemented in a pending IEEE standard 1596 for communication in a shared memory multiprocessor, the Scalable Coherent Interface (SCI). The SCI synchronization primitives are based on QOLB (originally called QOSB), a hardware primitive previously presented at ASPLOS-III, that shows much promise for reducing and/or eliminating the latencies associated with synchronizing on and accessing shared data. Introducing finer-grained programs in the absence of such latency reduction will likely have little or no benefit.

In particular, we discuss how QOLB fits the underlying linked-list cache coherence protocol of SCI. We also present and analyze two important scenarios showing that the QOLB primitives in SCI greatly reduce data communication latencies. These scenarios include critical sections, and the special case of pairwise-sharing. In addition, a number of other issues affecting correctness and performance are discussed, including cache line rollout, graceful initialization, process migration, and forward progress.

1 Introduction

One way to speed up the execution of a program is to distribute the computation over a number of processors. This approach is viable as long as most processors are doing useful work; that is, as long as they are not wasting time waiting for data to become local and ready for use. In shared-memory multiprocessors one important way of allowing data to reside locally is to provide caches, over which some measure of consistency is maintained.

As the execution of the program becomes more fine grained *i.e.* the number of instructions between points of communication is reduced), the sensitivity to access delays on shared data increases. Stone [Sto90] introduces the performance measure of MSYPS: Millions of SYnchronizations Per Second. If the amount of processing between synchronizations is comparable to the time to synchronize, then dividing the computation up further (making the granularity finer) will not produce commensurate speedup. Thus the speed of synchronization, or MSYPS, is a fundamental limit on the granularity of parallel programs, and therefore a limit on the number of processors that can cooperate on a single task.

We expect that future processors will be fast enough so that, even considering only speed-of-light limits, interconnection delays may be the equivalent of many instruction-issue times. Programs running on such cooperating processors will be limited in their parallelism by the time to synchronize on and exchange data. This limit can be ameliorated to a large extent by *prefetching* data, at least for networks that provide sufficient bandwidth and programs that are able to request data in advance [MG91]. But latency in accessing shared data cannot be entirely hidden – data written by another processor cannot be fetched before it is generated. Thus, multiprocessors need to provide efficient mechanisms for the acquisition of shared data. Such mechanisms must be effective for those paradigms of sharing most likely to be found in parallel applications. In this paper we look at synchronization and data exchange scenarios that we find to be important in parallel computing, and we analyze the performance of our hardware mechanisms and compare them to previous proposals.

One important scenario is this: In most applications access to a shared variable is restricted at any particular time to one of two cases. Either a process has exclusive read and write access to the variable, or one or more processes have read (but not write) access to the variable. A very common special case of sharing occurs when exactly two processes take turns reading and writing shared data. This situation is called *"pairwise sharing"* . In shared-memory multiprocessors the one-writer/many-readers paradigm of sharing may be enforced by a cache coherence protocol. Archibald and Baer describe and evaluate a number of protocols with this property [AB86].

A limitation with coherence protocols is that they only guarantee consistency for a single writable entity, and only for the duration of a single write operation. Most parallel applications require exclusive access to sets of related variables during a number of consecutive read and write operations. Enforcement of this sharing mechanism, called a *"critical section"* , generally requires special synchronization operations for efficient implementation. Many shared-memory multiprocessors use locks to enforce access to critical sections. Special atomic synchronization primitives such as *"Test&Set"* and *"Unset"* can be provided to acquire and release locks.

In addition to critical sections, there are numerous other paradigms for synchronizing processes and sharing data. In barrier synchronization [Jor83] a number of processes may wish to guarantee that all have reached a specific point in their execution before any can proceed. As a second example, processes may wish to perform enqueue and dequeue operations in parallel on a queue whose entries represent separate units of work. Although numerous scenarios exist, it is arguable that only a very few of these would warrant special hardware support. Even for barrier synchronization, which is a primary candidate for hardware support, efficient software solutions exist [YTL87]. In this paper, we restrict our attention to efficient support for critical sections.

The accessing of a critical section protected, for example, by a lock, results in at least three distinct latencies. First, a process must wait for the critical section to become free (*i.e.* the lock is unset). Second, the operation to acquire the lock requires time to traverse the interconnect. Finally, the process must issue and wait for requests that make the shared data local to the processor where the process is executing. As

the granularity of sharing becomes finer, these delays may dominate the time to complete a task.

One mechanism that shows promise for reducing and/or eliminating all three of these latencies is the QOLB primitive [GVW89]. In this paper we explore hardware support for locks, critical sections and data exchange using QOLB. Even though software solutions exist for this type of synchronization [MCS91, And90], one of the main points of this paper is that there is a significant benefit in having such hardware support for reducing memory latency.

Specifically we discuss the implementation of QOLB in a shared-memory multiprocessor using the Scalable Coherent Interface (SCI). SCI is a pending standard, designated as IEEE Standards Working Group P1596 [IEE91]. SCI is designed to provide an efficient, cache-coherent, shared-memory model to a large number of processing nodes. Due to the similarity between QOLB and the SCI implementation of cache coherence, it is natural to extend that implementation to include QOLB, and such an extension is provided as an option to the base SCI protocol. This paper also discusses and analyzes this extension.

Previous work [GVW89] has described the use of QOLB primarily for eliminating contention over the interconnect. The present work discusses QOLB's ability to reduce memory latency by (1) making synchronization common operations more efficient through the elimination of most traversals of the interconnect and (2) by allowing shared data to be prefetched, especially in how this relates to cooperating processes. Similarly, previous discussions of SCI [JLG+90] do not focus on the implementation issues of QOLB or various performance enhancements, such as pairwise sharing.

The remainder of the paper is organized as follows. Section 2 presents work related to critical sections, software queues for locks and QOLB. It also provides a brief description of cache coherence operations in SCI. An extended discussion of QOLB hardware support and synchronization issues in SCI is presented in Section 3. Section 4 describes an implementation of QOLB in SCI. The benefits of using QOLB for a specific example, the producer/consumer problem, are analyzed in Section 5. Section 6 provides a more general discussion of critical section latency for QOLB and software queuing methods. Finally, conclusions are presented in section 7.

2 Previous Work

Critical sections are used to restrict access to certain shared data to at most one process at a time. New processes must wait at the entrance to the section if another process is presently inside. In general there are two forms of waiting: active waiting and passive waiting. In the former case a lock is associated with the critical section, and this lock is repeatedly tested by processes wishing to acquire mutually exclusive access. In the passive waiting case there is a queue of waiting processes. Among other things, the advantage of a queue is that a process or processor could do useful work while waiting, and no redundant messages will be generated across the interconnect. Another advantage is that a queue can easily be made fair: once inside the queue, a process will not be overtaken indefinitely by other processes. On the other hand there is the overhead of inserting and removing processes from the queue.

Historically, *Test&Set* operations have been used to spin-wait actively, while semaphores have been used to wait passively. Since semaphores are shared variables, locks have usually been used to implement the critical sections of the semaphore operations. Hence most computer instruction sets include an atomic lock operation in order to efficiently implement wait queues. In many systems simple lock operations are the only hardware support available for implementing critical sections.

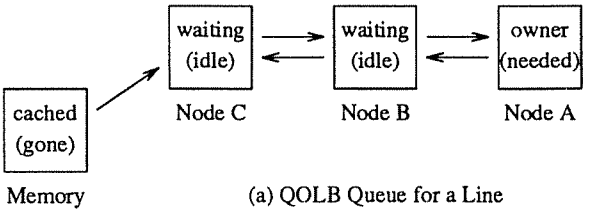
In a shared memory multiprocessor, resources can be wasted if critical sections are not implemented carefully. Additional latencies in the critical path of an algorithm and unproductive use of interconnect bandwidth, are the result of naive synchronization mechanisms for critical sections.

In multiprocessors, the *Test&Set* operation may give better performance if implemented instead as a *Test&Test&Set* operation [RS84]. This primitive allows processes to spin wait on a locally cached copy of a lock. When the lock is unset all shared copies are invalidated, after which exactly one process will succeed in setting the lock. Unfortunately, while *Test&Test&Set* reduces spinning over the interconnect, contention for a lock may result in unfair allocation and large amounts of interconnect traffic if the lock is held only

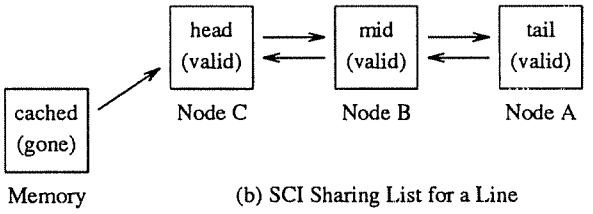
momentarily.

2.1 QOLB and Hardware Queues

The QOLB primitive [GVW89] was designed to provide efficient hardware support for critical sections by allowing processes to build distributed hardware queues of waiters for cache lines. [1] A line is the memory entity across which coherency is maintained. By providing a direct implementation of a binary semaphore queue, QOLB can be used as a mechanism for efficient process synchronization. As a non-blocking operation QOLB can prefetch *i.e.* make local) a line of data while a process performs useful work. Combining these two operations along with a simple software convention, QOLB becomes a *"synchronizing prefetch"* operation. That is, QOLB can be used to synchronize and fetch a line of data, allowing local tests to determine when the data has become available. For example, a line of data would migrate to the next process in the QOLB queue when the *owner* process (the process holding the lock) releases the line. If prefetched sufficiently in advance, a process could synchronize on and access shared data without experiencing delays from transactions over the interconnect.



(a) QOLB Queue for a Line



(b) SCI Sharing List for a Line

Figure 1: Similarity between SCI and QOLB Queues

A QOLB queue is shown in figure 1. Node A, called the owner, has the only valid copy of the line. Processors waiting (Node B and C) for the line have stale, or shadow, copies on which local processes may spin. In SCI the queue is doubly linked, with memory keeping only a pointer to the head of the queue (every node has an implicit pointer to memory). The SCI cache states are listed in parenthesis and will be explained in more detail in the sequel.

The QOLB primitive can be used to define three operations: `acquire(line)`, `release(line)` and `prefetch(line)`. The precise implementation and semantics of these operations depend on the exact implementation. An example is given in Section 4. The acquire operation adds an entry for the requesting processor into the queue for the line. The release operation causes an owned line to migrate to the next processor in the queue, which then becomes the new owner. The prefetch operation attempts to make a

[1] The acronym QOLB, pronounced *"Colby"*, (formerly called QOSB) stands for *"Queue On Lock Bit"* .

line local, while freeing the processor to continue execution. By placing a lock in the desired line, and using *Test&Set* and Unset to acquire and release the lock, the above operations allow a process to enter, exit, and even prefetch the lock and data for a critical section. A lock is needed in addition to the cache-states defining the ownership, because a line may be rolled out from any cache at unpredictable times.

Two software algorithms inspired by QOLB have been developed to minimize network contention for a lock. Each of these implements queues as software-maintained data structures. Anderson [And90] presents a scheme to implement a queue as a circular array. Mellor-Crummey and Scott [MCS91], on the other hand, chose to implement a queue as a linked list. Both algorithms succeed in reducing traffic across the interconnect to a constant number of traversals per lock access. However, neither allows a lock, or the data associated with a critical section, to be easily prefetched. A comparison of QOLB and the software queue techniques in the context of executing critical sections is presented in section 6.

2.2 Scalable Coherent Interface (SCI)

SCI is a pending IEEE standard that defines the physical and logical interfaces between modules (called nodes) in a shared-memory multiprocessor [JLG+90]. An SCI multiprocessor can contain up to 64K nodes, each node containing one or more processing elements (with cache), a memory module, a DMA adapter, or a combination of these. Communication between nodes is based upon the sending of messages of two types, requests and responses. A low-level logical layer defines arbitration and flow control such that messages are transported reliably and within a predefined time from a source node to a destination node.

SCI defines a chained-directory-based, cache-coherence protocol [IEE91]. Memory that can be coherently cached is divided into lines that are 64 bytes long. For each line there is a distributed directory that defines the set of nodes whose cache contains a copy of the line. The directory is implemented as a doubly-linked list of cache lines, called a *"sharing list"* . For each memory line there is state information and a pointer to the first cache line (the head) in the sharing list. For each cache line there is a forward and a backward pointer. In addition, cache lines contain state information describing the cached data as well as the position in the sharing list (head, mid, or tail). A sharing list is shown in figure 1, with the SCI cache states listed for each node.

The close correspondence between hardware queues and sharing lists makes SCI attractive for implementing QOLB. A major difference is that in a QOLB queue only one node has a valid copy of the line. All other copies are *stale* , although the QOLB implementation allows processes to test the state of their local copy to determine if the data has been made local.

In a sharing list, when there exist more than one valid copy of a memory line, it is legal to read but not write the contents of these copies. New readers and writers are prepended to the head of the sharing list by first going to memory, and then to the old head. The two request/response pairs needed in this case are in principle shown in the two first pictures of figure 2. In order for one copy to be modifiable it must be the only existing valid copy, including the copy in memory. The state of such a line is called *exclusive* . The SCI cache coherence protocol is based on write invalidation. A head line can become exclusive by invalidating the rest of the sharing list.

Figure 2 shows the steps involved in acquiring exclusive access to a line, specifically a line previously in exclusive mode. First a processor (node C) interrogates the directory for the line in main memory to determine the current head (node B). Second, the processor acquires the line from the head. If the processor needed a read only copy, then the previous head would have been set to state *valid* . The case shown, however, is one where a writable copy is needed, so the old head is marked stale and the new head exclusive. Finally, the processor invalidates the rest of the sharing list, which in this case consists of a single processor (node A). In general, if a new writer appends to a long (read) sharing list, this last step may require a series of invalidations.

SCI has optimized the important special case where exactly two processors share a copy of a line. Then the tail may acquire read and write privileges as well. The other (invalid) copy in a two element list is in the state stale. This allows for efficient pairwise sharing of a line by allowing the head and tail to read and write the most current copy of the line without ever accessing the memory directory.

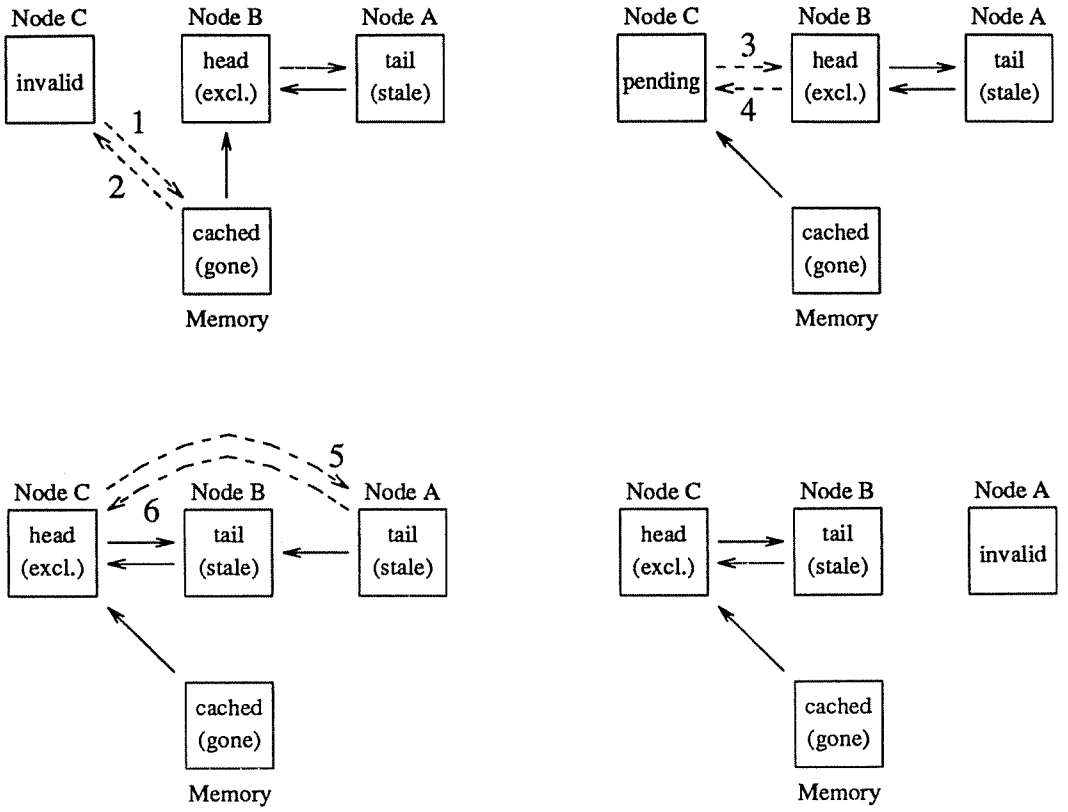


Figure 2: Acquiring Exclusive Access to a Line from a Previous Writer

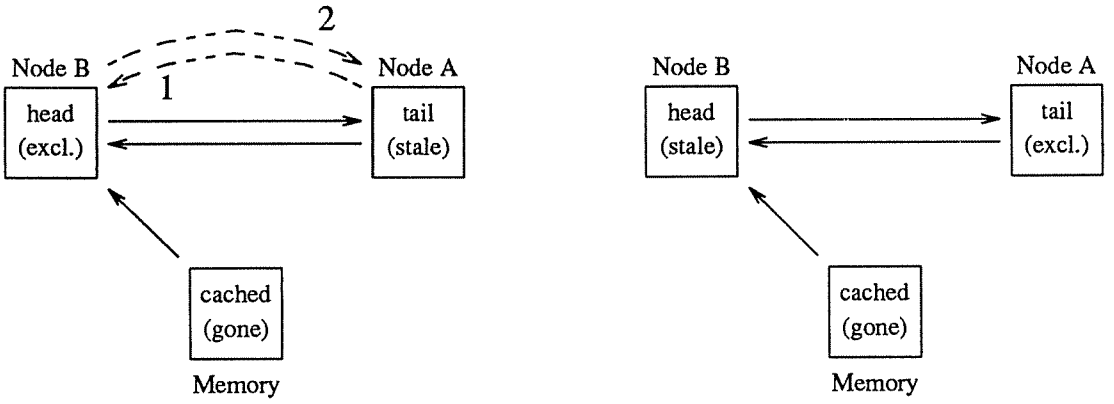


Figure 3: Acquiring Exclusive Access Using Pairwise Sharing

Figure 3 shows the steps involved in acquiring exclusive access to a line when in a pairwise sharing situation. In this case, node A (rather than a new node C) is attempting to acquire exclusive access to a line that is held by node B. Only one pair of messages is involved in this transaction, as compared to the multiple messages shown in figure 2. Thus, a substantial savings in terms of both bandwidth and latency is possible.

3 SCI Implementation Issues for Supporting QOLB

This section discusses important issues in the design of a QOLB primitive as part of the general SCI protocol. These issues include the interaction of QOLB with other aspects of the SCI protocol, initialization of memory, cache line roll out, process migration, and forward progress. Section 4 summarizes the semantics of the QOLB primitives. A protocol description and implementation of QOLB in SCI can be found in the proposed SCI standards document [IEE91].

3.1 Including QOLB Functionality in the SCI Protocol

The general SCI cache coherence protocol defines a sharing list of nodes holding readable copies of a line. Additional nodes may *prepend* to the list by inserting themselves at the *head* of the list and acquiring a copy of the data from the old head. Only the head node may write the line, and then only after invalidating or purging all the other nodes, collapsing the list to a single node. Several nodes may prepend simultaneously, being serialized by the memory controller.

The QOLB protocol has a similar structure, where nodes join a queue in FIFO order, and wait until they arrive at the other end of the queue. The node that reaches this other end is the one that has waited the longest in the queue. It is this node that gets exclusive access to the data. The primary difference between a QOLB list and an SCI sharing list, is that an SCI list consists of readers only. In the case of a writer prepending to a sharing list, the list is broken down in order for the writer to become the exclusive member. A node prepending to a QOLB list, on the other hand, is willing to wait with its task of writing until the other nodes in the queue have completed their work. Hence waiting times are controlled by software.

When implemented in SCI, the QOLB queue is handled in a manner similar to a read sharing list. The node having exclusive access to the line is called the *"QOLB owner"*, and is the tail of the sharing list. A node wishing to join the QOLB queue prepends in a fashion similar to a reader to a conventional sharing list, and becomes the head of the list. However, it does not receive ownership until all nodes that previously joined the queue have acquired the line and voluntarily relinquished it. When a node relinquishes QOLB ownership of a line, it also deletes itself from the list.

The original QOLB proposal specified that the queuing operation, including the associated lock operation, be performed on an extra bit (the lock bit) associated with each line. For a cached line, this information need not be stored explicitly, since it can be inferred from the cache line state. However, it must be stored explicitly for lines that are rolled out of a cache – either to main memory or to disk. In order to avoid the complexities of handling out-of-band lock bits, SCI permits in-band lock bits. For a system using in-band lock bits, a conventional synchronization operation, such as *Test&Set*, can be used for locking the memory line. A lock bit must be used if the process wants to ensure that the data in the line is protected from use by other processes.

An UnQOLB operation is provided to indicate the termination of exclusive access. This operation passes the exclusive copy of the line to the next waiting node (if any) along with QOLB ownership, and deletes the relinquishing node from the list. Normally preceding the UnQOLB operation is an Unset operation to the lock bit, allowing the *Test&Set* operation to succeed when the line arrives at the new QOLB owner. If there is no waiting node, the UnQOLB has the effect of changing the state of the line so that an ensuing QOLB will forward the line immediately. An UnQOLB operation may also be performed by a node having previously issued a QOLB but now deciding not to wait. In this case it is likely that the node is in the QOLB queue, and the effect is to remove the node from the queue. An UnQOLB operation issued by a node

not in the sharing list, or a part of a conventional SCI sharing list is normally ignored. [2]

A new operation, called ReQOLB, has been included in the SCI implementation of QOLB. This operation is performed by a processor that wants to send the shared line to the next processor in the queue, and at the same time rejoin the queue. In this way the line will come back as soon as all the other waiters have completed their accesses. Hence ReQOLB is an atomic, sequential combination of the operations UnQOLB and QOLB. The semantics of ReQOLB can be seen in the state transition diagram in figure 4, by combining of transitions for UnQOLB and QOLB. When there are no other processors waiting for the line, the ReQOLB operation is a no-op. This operation has been included because it can reduce the number of internode communications by 50case. This case occurs when two processors alternately access a line.

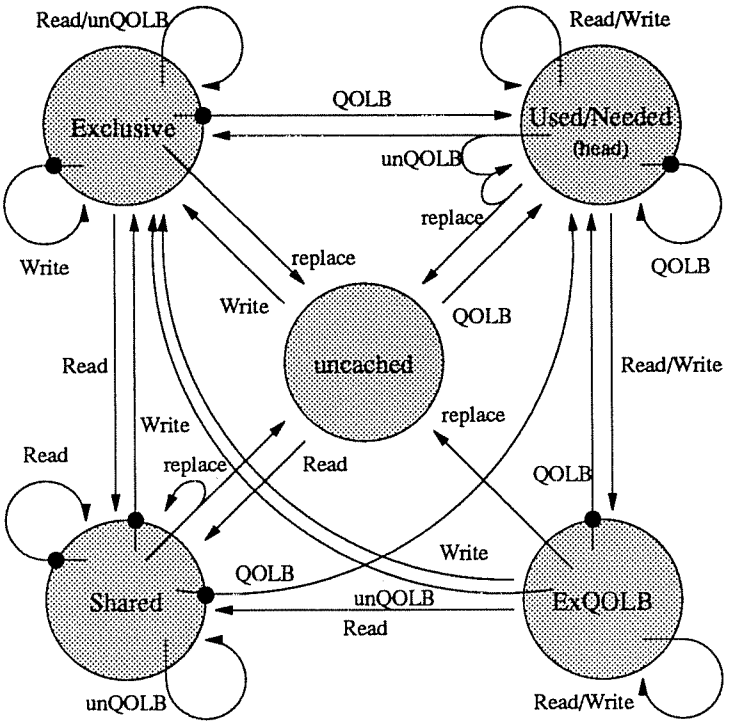


Figure 4: Some Important SCI-QOLB Sharing Lists and Transitions

Finally, the QOLB protocol is an option in the SCI protocol, and may not be implemented in all systems. QOLB and UnQOLB are defined in a way that nodes not understanding the QOLB protocol can simply ignore these operations. In general, any node can ignore a QOLB operation at any time. The resulting effect is only a loss of performance – correctness is guaranteed by proper use of the lock bit. We implicitly add this constraint to the following discussion of QOLB implementation issues. Note however that when QOLB is used, the lock and the rest of the shared data reside in the same memory line. When other synchronization schemes are used, it is probably wise to allocate separate memory lines for the lock, which is read-shared, and the data, which is being written.

[2] But see later discussion of process migration.

3.2 Initialization of Memory

The QOLB option allows the memory to build up additional state information in the form of a queue of processes waiting to access a line. A protocol is defined for the use of the QOLB primitives so that the data will be shared in a predictable and controlled manner. However, the queues are built up by programs that may be incorrect, or may be terminated in unexpected ways, leaving behind QOLB structures to be inherited by future programs on startup. If a program uses QOLB, it is reasonable to expect it to go through an initialization process, just as one initializes any kind of data. But if the program does not use QOLB, it should not be required to use a special mechanism for initialization. It is imperative, therefore, that simple reads and writes being issued in any order from any collection of nodes must reduce the QOLB structures quickly and gracefully to conventional SCI structures. This property is referred to as *"graceful initialization."*

Programs that intend to use QOLB operations must properly initialize memory before it can be used safely. This function may be provided as a system or library call, and is required only as initialization for regions of memory in which QOLB operations are performed. Other parts of memory are initialized simply with reads and writes. The initialization of a line for QOLB use can be done by performing an UnQOLB operation followed by a write, and then another UnQOLB. The first UnQOLB is needed to ensure that the initializing node is not the QOLB owner, or present in the QOLB queue. The write operation from a node not in the QOLB queue breaks down the QOLB queue, if any. This action is similar to the process of invalidating a sharing list, except that the node at the end of the list has the only valid copy of the data. The final UnQOLB is necessary for reasons explained in the next section.

3.3 Cache Line Rollout

The SCI philosophy prohibits a line from being locked into a cache indefinitely. Since a QOLB queue may remain in a given state for an arbitrary period, provisions must be made for any line to be evicted from a cache. Generally, this situation is easily handled, in the same way that a line in a conventional sharing list may delete itself from the list. While a node is waiting for a QOLB line, it spins locally on a shadow copy of the line. When the line is rolled out, this shadow copy disappears, but since the node was still waiting to receive the resource, it will continue generating QOLB requests. The next QOLB request will put the node back in the QOLB queue, and a new shadow copy will be created for local spinning. There may be some undesirable effects such as loss of position in the queue, as well as some extra message traffic generated when the node rejoins the QOLB queue. If this happens more than occasionally, it suggests an unbalanced system, since an actively accessed cache line normally should not be rolled out. A later section discusses this and other issues of fairness.

When the line of the QOLB owner is rolled out, however, the line is transferred to the next node waiting for it, which becomes the new QOLB owner. A QOLB operation executed on this node will now succeed. Were QOLB the only synchronization mechanism, the protocol would break, since a process on the next waiting node might now enter a critical section without the process on the old QOLB owner having completed its critical section. However, the lock bit is left set, so while the QOLB succeeds, the *Test&Set* operation will fail. Thus, even though this node has received the line, it will continue spinning locally on the lock. (The same situation obtains if two processes on one processor attempt to concurrently access a critical region). The subtle part is how to get the lock holder back as the QOLB owner. The lock holder's next access to the line may be either a write, a read or an UnQOLB, depending on where it was in its critical section.

If the next access is an UnQOLB, the operation is ignored, as explained above. If the next access is a read or a write operation, then the QOLB queue is broken down, and the cache (of the lock holder) receives the line in an exclusive state. This state, called EXQOLB, is used to record the fact that it resulted from the break down of a QOLB queue. (EXQOLB accounts for the additional UnQOLB required in the initialization procedure, and is explained more thoroughly below). Subsequent read and write operations in the same cache will not change this state. However, read and write operations from other processors must remove the QOLB information and revert the list from EXQOLB to a normal sharing list in order to provide for graceful initialization. If a QOLB request finds a line in the EXQOLB state, the line is simply changed to a QOLB state, instead of forwarding the line to the requester, as when the line state is EXCLUSIVE. This

prevents a QOLB queue from repeatedly being built up by QOLB operations from waiting processes and broken down by reads and writes from the lock holder, as might occur in the case where a QOLB line was rolled out of the lock holder's cache.

The state and transitions described above are shown in figure 4. Circles represent the possible stable lists, in addition to the uncached memory state. A transition starting from inside a circle indicates that a node in that list caused the transition. A transition starting from outside a circle indicates that a node not currently in that list caused the transition. Note that some transitions result in the same type of list, but one with different elements. For simplicity, some transitions not pertinent to QOLB have been left out.

3.4 Process Migration

Another issue that must be handled carefully is process migration. While most problems of process migration can be handled by the same mechanism that ensures proper behavior in the case of a cache line rollout, one additional consideration requires special handling. This problem arises from the fact that QOLB operations are associated with a physical node, and not with a process.

There are two aspects to that problem. The first is how to ensure that a process that is migrated while within a critical section will properly relinquish QOLB ownership. The second is how to ensure that a process that is migrated while waiting to acquire exclusivity does not leave behind a QOLB entry on a node without a process waiting to accept and release QOLB ownership. Both problems arise because a process entering a QOLB queue creates a situation that requires explicit (software) actions to resolve. In the first case, the QOLB owner must be explicitly released so that ownership can be passed to other nodes. In the second case, an entry in the QOLB queue must have an attendant process that either removes the node from the queue before it becomes the owner, or that releases ownership after it has been acquired. Thus the QOLB operation carries with it an obligation to ensure the release of resources. The UnQOLB operation carries no such obligation, and is always safe, that is, any node may issue an UnQOLB operation at any time without introducing the possibility of incorrect behavior (though excessive use of this ability may cause starvation).

The first problem can be solved by guaranteeing that the QOLB owner is always deleted from the QOLB list. In the normal case, of course, the QOLB owner is the node issuing the UnQOLB, but when a node is migrated while in a critical section, the ensuing UnQOLB may be issued from a different node. Some procedure is necessary to ensure that the node from which the process was migrated does not remain the QOLB owner. A sufficient mechanism would be to define UnQOLB in a way that one QOLB owner is always removed. One way to ensure that this happens is to break down the queue whenever an UnQOLB does not succeed locally. This definition is obviously undesirable, and better solutions are under investigation.

The second problem presents a different dilemma. Each time a node issues a QOLB operation, it will create a waiting node in the QOLB queue if one does not already exist. If it is migrated, it will create such a node in the QOLB queue, and some mechanism must be employed to purge the node from which the process was migrated.

Two possible mechanisms have been identified to guarantee correct behavior in the presence of process migration. One depends on either the process or the operating system to UnQOLB all lines that might possibly have an outstanding QOLB operation before migrating a process from a processor. This solution is not attractive, since it implies either flushing the cache (at least the address space of the migrating process) or requiring the process to maintain a list of all lines that might possibly be in QOLB queues. A second solution involves some hardware support: the QOLB and UnQOLB operations return a value to the user, namely, the node of the ID where the operation was performed. In addition, the UnQOLB operation takes a node ID as a parameter, and the UnQOLB operation can be performed remotely if required. This mechanism appears promising, and is currently under study.

3.5 Fairness and Forward Progress

Synchronization primitives are expected to provide mechanisms that allow a variety of policies to ensure fairness, as well as forward progress. First-come, first-served access is provided naturally by QOLB, so that

basic notions of fairness are easy to implement. However, perfect ordering is not guaranteed because of the need to break down the QOLB queues under certain circumstances. QOLB queues can be broken down as a result of only two actions: (1) a cache line rollout, and (2) a process migration. Because QOLB entries are assigned to nodes rather than processes, processes sharing the same node also share a common position in a QOLB queue. Except for these circumstances, however, first-come, first-served ordering is provided.

Cache line rollout can generally be prevented for a reasonable critical section on a carefully designed system. Process migration likely cannot be controlled by the user; however, it is usually infrequent. If forward progress must be guaranteed, a program can still use the QOLB/UnQOLB primitives, monitoring its behavior appropriately, and switch to less efficient, software synchronization techniques [And90, MCS91] to guarantee progress. Since such behavior is expected to happen relatively rarely, this solution will not impair performance in any appreciable way. All that is needed is a counter in the outer loop to monitor progress and determine when changing algorithms is necessary.

4 The SCI QOLB Implementation

In order to implement the QOLB protocol, four extra cache states are needed in the SCI protocol. Three of these states are called USED, NEEDED and IDLE. A line in state USED or NEEDED implies that the node is the QOLB owner of the line. NEEDED additionally indicates that there are nodes waiting to become the owner. The shadow lines of such nodes are denoted by the state IDLE. Lines in states USED and NEEDED are readable and writable, whereas lines in state IDLE are unreadable and unwritable. The fourth state, called EXQOLB, is used to annotate the state EXCLUSIVE with the information that it resulted from breaking down a QOLB queue. This was described in the discussion of cache line rollout.

The implementation of the different QOLB operations is given below. We assume that processes are not migrated. This implementation defines QOLB slightly differently from previous work [GVW89, WG91].

QOLB(line).

A QOLB operation is implemented as a special read operation on the specified line. The operation returns a boolean value: success or failure (respectively 0 and 1). This value is returned at once after a lookup in the cache. Success is returned only if there is a cached copy of the line, and this line is in state USED, NEEDED, or EXQOLB. In all other cases, including a cache miss, the returned value is failure. If a QOLB operation is issued on a memory line with a sharing list where the head is not in one of the four QOLB states, the sharing list is broken down, and the requester receives a copy in state USED. If another node issues a QOLB operation on the line, then it joins the list in state IDLE, and the node that previously was in state USED is changed to state NEEDED. New lines that are added after this, will simply join the queue in state IDLE. The QOLB owner, which is in state NEEDED, is not affected by these new additions.

UnQOLB(line).

When the QOLB owner executes an UnQOLB operation the exclusive copy of the memory line is sent to the node that will become the new tail, which then changes from state IDLE to state NEEDED, and the old tail is invalidated. If the new tail is the only remaining node, it enters state USED instead of state NEEDED. If, after an UnQOLB operation, there are no other nodes in the list, then state is simply changed from USED or EXQOLB to EXCLUSIVE.

If a processor decides not to wait for a critical section any longer, it executes an UnQOLB operation which rolls the IDLE cache line out of the QOLB queue. If the process executing the UnQOLB operation gets a cache miss, or finds the line cached in a non-QOLB state, the UnQOLB operation is a no-op. [3]

[3] Hardware support for process migration might dictate somewhat different semantics.

ReQOLB(line).

This operation is implemented as a combined and atomic execution of UnQOLB and QOLB. A QOLB owner in state USED remains in this state as a result of the execution of an ReQOLB operation. When the owner is in state NEEDED, the new tail changes from state IDLE to NEEDED, and the line reenters the sharing list as a head in state IDLE. Note that this operation is not used in the library functions defined below. ReQOLB is most useful when there is a fixed set of processors cyclicly sharing a resource, as will be demonstrated in section 5.

Read and Write operations.

The cache states NEEDED, USED and EXQOLB are both readable and writable states. All other read and write operations break down the QOLB list and leave the node with a cache line in the state EXQOLB.

In light of the above definitions of QOLB and UnQOLB, the acquire, release, prefetch, and initialize operations can now be specified. These three operations should be library functions that users call in order to use QOLB correctly. The acquire operation is a modified spin-lock in which a process spins on the line as long as the QOLB operation returns failure. When the line has become local, a *Test&Set* operation is used to determine the state of the lock, which would be expected to be unset except in exceptional circumstances. If it is locked, then the QOLB queue must have broken down (or will soon), and the process must continue to spin on the line using QOLB. Note that a fallback scheme, necessary to guarantee forward progress, is not included here.

```
acquire(line)
{
     do
          while (QOLB(line)) do
               ; /* spin-wait */
     while (Test&Set(line.lock))
}
```

The release operation simply unsets the lock and UnQOLB's the line, allowing the next node in the queue to become the owner. Since the QOLB operation does not block the process while it waits for a line it can also be used alone as a prefetch operation, as shown below. The last operation, initialize, has been described in the previous section.

```
release(line)
{
     Unset(line.lock)
     UnQOLB(line)
}

prefetch(line)
{
     QOLB(line)
}

initialize(line)
{
     UnQOLB(line)
     Unset(line.lock)
     UnQOLB(line)
}
```

4.1 Pairwise Sharing Using QOLB

Both the pairwise sharing and QOLB features of SCI were intended to support efficient sharing of data. Pairwise sharing was incorporated specifically in recognition of the importance of efficient sharing between pairs of processors in fine-grained parallel applications. Both QOLB and pairwise sharing reduce the number of network traversals, thereby increasing the efficiency with which individual operations can be accomplished.

To enable the QOLB primitives to exploit pairwise sharing support, the obvious extension to QOLB is to allow the head of a two element list to be the QOLB owner. The other element in the list is then either in state IDLE (if it wants the line), or in state STALE (in case it has not indicated interest in acquiring the line). This alternative implementation of the QOLB queue is also supported in the SCI protocol.

As an example of the interaction between QOLB and pairwise sharing mode, consider the case where two nodes alternately enter a critical section to read and write a shared line of data. When the line is first acquired by one of the nodes using QOLB, the resulting list will contain only that node, with the line in state USED. When the second node performs a QOLB on the same line, it joins as the head of the sharing list in state IDLE, and the tail node switches to state NEEDED. When the line is released by the tail, a single cache-to-cache write is needed to transfer the contents of the line to the other node, leaving the head in state USED and the tail in state STALE. If the tail node now issues another QOLB, it simply sends a message to the head node, which then changes from USED to NEEDED. The tail itself changes from STALE to IDLE. When the head has completed its critical section, it performs a cache-to-cache write, going from NEEDED to STALE, and the tail changes from IDLE to USED, etc. If both nodes are executing outside the critical section, *i.e.* neither has performed a QOLB) then one is in state STALE and the other in state EXCLUSIVE. The first to execute a QOLB operation will get the ownership (and the data) and leave the other in state STALE.

A node pair in state (NEEDED,IDLE) will go directly to the mirror state (IDLE,NEEDED) if the QOLB owner executes a ReQOLB operation, thus avoiding the state STALE and the extra transmission it takes to go from the pair (STALE,USED) to (IDLE,NEEDED).

If a third node tries to join the queue, then pairwise sharing mode is dropped. If at that time the head node is in state NEEDED and the tail is in state IDLE, the tail is simply purged. The alternative would be to restructure the list, which is less efficient in terms of the number of messages that have to be sent across the interconnect. If the tail is dropped, then it simply rejoins the queue when it issues its next QOLB operation for that line. If one of the two sharers are in state STALE, then this node is simply dropped from the list. The transition between the normal list mode and pairwise sharing is so efficient that pairwise sharing is in effect whenever the list has exactly two nodes.

5 Analysis of the Producer/Consumer Problem

The producer/consumer problem is a very important example of pairwise sharing. In this section we describe a common scenario for the producer/consumer model, and show how QOLB makes it possible to exploit this synchronization method in fine-grained parallel execution.

The scenario is as follows: Producer P1 is alternately writing to buffer A and buffer B. Consumer P2 is alternately reading buffer A and B after P1 writes them. Synchronization must guarantee that the two processors proceed in lock-step, that is, P1 must not get so far ahead that it is acquiring a buffer it has produced but is not yet consumed. P2 must never catch up, that is, it must not consume buffers that have not yet been produced. In the model, P2 is not restricted from writing the buffer as well, providing for symmetry in the operation if desired. This provides a more powerful programming model than the pure producer/consumer model.

Each processor alternates between acquiring resources to enter a critical section and executing within a critical section. As soon as P1 reaches the end of a critical section it releases its lock, then immediately attempts to acquire the other lock. The following segment of code shows how a conventional producer/consumer exchange might occur.

```
while (1) {
    do {
        while (Test&Set (A_lock))
          ; /* spin-wait till I acquire the lock */
    } while (A.owner != me);

    Use_Structure (A); /* critical section */

    A.owner = (me+1) % 2; /* pass the buffer on */
    Unset (A_lock);

    do {
        while (Test&Set (B_lock))
          ; /* spin-wait till I acquire the lock */
    } while (B.owner != me);

    Use_Structure (B); /* critical section */

    B.owner = (me+1) % 2; /* pass the buffer on */
    Unset (B_lock);
}
```

Processor P1 begins execution with me=0, while processor P2 begins execution with me=1. Note that A_lock should be in a different cache line than the rest of the data structure A, since the lock will migrate to the processor not holding the lock. The owner identification, A.owner, could reside in the same cache line as A (and vice versa for B_lock and B.owner). Now consider the sequence of events that must occur when P1 tries to acquire B at the same time that P2 tries to acquire A.

1. P1 unsets A_lock and attempts to set B_lock, requiring a round-trip network delay to acquire B_lock. (In the worst case, this might require multiple round-trips, since B_lock might be fetched before it is released by P2, in which case P2 would have to refetch it to release it.

2. P1 succeeds in acquiring B_lock, and now attempts to access the guarded data. Assuming that all the data (except the lock) are contained in a single cache line, an additional round-trip delay is required to access the data, which is resident in the cache of P2.

Thus four network traversals delays are necessary from the time a processor initiates its attempt to acquire the lock and the time at which it may access the guarded data. The situation is even worse if P1 and P2 differ significantly in the time they spend in the critical section. If P2 tries to acquire A_lock before P1 releases it, P2 fetches A_lock into its cache but cannot succeed. Now when P1 tries to release the lock, it must first invalidate the copy in P2 before releasing the lock. Upon invalidation of its copy, P2 must refetch a copy of A_lock from P1. Only after releasing A_lock can P1 fetch B_lock and acquire the lock. Thus in this case the processor running behind, P1, must wait for six network traversals before it may enter the critical section. This delay can be reduced to four by recognizing that P1 need not wait for the unsetting of A_lock to complete before attempting to acquire B_lock. Thus a relaxed consistency model such as release consistency [GLL+90], can limit the waiting time to only four network traversals.

Consider now how QOLB can reduce the waiting time. Synchronization time can be mostly or entirely overlapped with computation. The owner identifications (A.owner and B.owner) are allocated in the same cache as the rest of the data.

```
QOLB (A);
QOLB (B);
while (1) {
    while (1) {
        while (QOLB (A))
        { /* Check for B -- should be gone now */
            if ((! QOLB (B)) && (B.owner != me))
               ReQOLB (B); /* I've already seen this buffer */
        }
        if (A.owner == me) break;
        ReQOLB (A); /* I'm too far ahead */
    }

    Use_structure (A); /* critical section */

    A.owner = (me+1) % 2;
    ReQOLB (A);

    while (1) {
        while (QOLB (B))
        { /* Check for A -- should be gone now */
            if ((! QOLB (A)) && (A.owner != me))
               ReQOLB (A); /* I've already seen this buffer */
        }
        if (B.owner == me) break;
        ReQOLB (B); /* I'm too far ahead */
    }

    Use_structure (B); /* critical section */

    B.owner = (me+1) % 2;
    ReQOLB (B);
}
```

After the first iteration, the QOLB operation is initiated in advance, allowing the request to be overlapped with computation. Note that both processors continuously have QOLB requests pending for both lines, releasing a line only momentarily to explicitly allow it to migrate to the other processor's cache. Note also that the two internal infinite loops (that are exited through break statements) are specialized acquire functions that should preferably be hidden in a well documented library. This code exhibits the following features:

1. When a processor exits its critical section, it releases the line with a ReQOLB operation. After a single network traversal delay, the buffer has migrated to the other processor. If the two processors are operating in lockstep, only a single network traversal delay is lost due to synchronization delays. This demonstrates the prefetching capabilities of QOLB.

2. If one of the processors is running ahead, it may have to wait; but for the one running behind, the network latency in its critical path can be overlapped with processing. For example, if P1 is not able to keep ahead of P2, which completes its critical section sufficiently far in advance, P2 will spin-wait for a buffer to arrive from P1. However, P1 will find its buffer has already arrived when it exits a critical section and is ready to acquire one. Figure 5 shows a timeline demonstrating how the two processes

naturally slip into phase so that the slower process never has to wait for its buffers to arrive, effectively experiencing zero synchronization latency, which is optimal for this case.

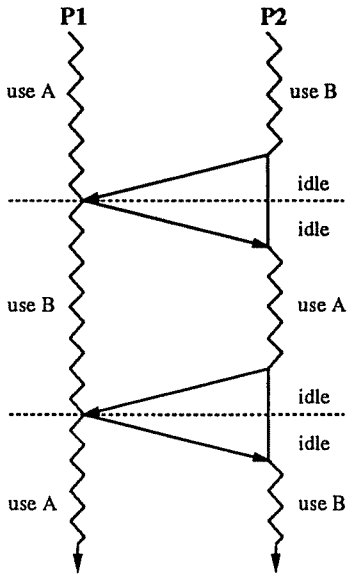


Figure 5: Producer/Consumer Timeline

In the pure producer/consumer model the producer can be given more opportunity to race ahead by providing additional buffers. The consumer, however, still must wait after attempting to acquire a buffer, so in the absence of prefetch, the network latency cannot be overlapped. In the QOLB example, the model can be fully symmetric, with each processor reading and writing each buffer, and the network latency does not lie on the critical path.

6 Comparison of QOLB with Software Queueing Methods

The previous discussion highlights the benefits of QOLB by illustrating its use for an important case of process synchronization, namely, the producer/consumer problem. In the present section we will discuss another very important synchronization and data exchange scenario. The mechanism under study is the critical section – a general mechanism for synchronizing the exchange of shared data between processes that is prevalent in many parallel applications. The objective in this section is to look at the performance benefits of QOLB in this broader context.

The main alternatives to QOLB are schemes that provide support for critical sections chiefly "in software." In actuality, all schemes analyzed here use atomic read-modify-write primitives. In the absence of contention, a simple spin-lock employing the *Test&Set* primitive is sufficient to guarantee mutual exclusion. However, if a number of processes are competing for entry to a critical section then providing for queues in software may be more appropriate. The latter methods include algorithms by Anderson [And90] and by Mellor-Crummey and Scott [MCS91]. For all of these algorithms access to a critical section is controlled by a lock.

In this section memory latency and interconnect bandwidth usage are compared for the SCI implementations of QOLB, Mellor-Crummy and Scott (MCS), and Anderson locks. For the case of pairwise sharing, where there is no contention for the lock, *Test&Set* spin-locks are also included. The main results are sum-

marized in table 1 at the end of the section. The reader is referred elsewhere for the details of the software queueing algorithms [And90, MCS91].

6.1 Critical Sections

A critical section consists of setting a lock, acquiring shared data, performing some computation, and releasing the lock. Any differences between synchronization mechanisms may show up in all but the computation portion. For fine-grained parallelism and with proper packing of data items, a single (cache) line of shared data may be all that is needed. For larger grained computations it is desirable that the latency of acquiring additional lines of shared data be overlapped with acquisition of the first line and/or with the critical section computation. Thus, for this discussion, only latency to acquire and release a lock and to acquire (*i.e.* make local) the first line of shared data are considered.

A process accessing a critical section may find the lock in one of two states, namely, idle or busy. Measuring idle lock performance yields information about the overhead of the locking mechanism, as well as the latency for acquiring free resources. In this situation it is important to include, not only the cost of entering the critical section, but of leaving as well, since the requesting process will experience both latencies. In the second case, a busy critical section, time is measured from the point at which the lock is released until the next process has acquired both the lock and the first line of data. This latency is a good indication of how long it takes for a resource to change hands.

The important special case of pairwise sharing occurs when exactly two processes alternately access a critical section. In the SCI cache coherence protocol pairwise sharing has been optimized by removing the need for either process to access the directory for the line being shared, as discussed in section 2. Since contention in a pairwise sharing situation is limited, a simple spin-waiting algorithm employing *Test&Set* can be used to provide the same or better performance than more complicated software schemes. For pairwise sharing, QOLB is compared against *Test&Set* as well as the Anderson and MCS schemes.

Test&Set, MCS, and Anderson's lock algorithms rely on *Fetch&*Φ operations to perform critical synchronization variable manipulations. Two types of *Fetch&*Φ implementations are possible, one using cache coherent line primitives with the *Fetch&*Φ operation performed locally, and the other using *remote word* operations with the *Fetch&*Φ operation performed by the memory controller where the shared variable resides. If it is assumed that an application can statically allocate variables so as to place them in the memories of specific processors, then spinning using remote word primitives may potentially be performed locally by a particular process. Both of these implementations of *Fetch&*Φ operations are considered.

6.2 Latency Calculations

Assuming that processors are relatively fast compared to memory and the interconnect, the time required to execute a critical section entry or exit procedure is dominated by the latency of the synchronization operations and accesses to shared data. To simplify the analysis it is also assumed that these operations consist solely of memory accesses and interconnect traversals.

Each processor is assumed to have two levels of cache, a fast first level (processor) cache and a slower second level cache. The latency of the processor cache is ignored. The large, second level cache is likely to be built using approximately the same speed DRAM as main memory. Hence, for simplicity, all memory accesses, including those to the second level cache, are assumed to require the same amount of time.

With the above restrictions the latency t for accessing a critical section can be computed for a given sequence of operations by the equation

$$t = n_{msg} * t_{msg} + n_{mem} * t_{mem}$$

where n_{msg} and n_{mem} are the number of one-way traversals of the interconnect by messages and the number of memory accesses, respectively, and t_{msg} and t_{mem} are the corresponding times for these events. Thus, calculating the latency of an algorithm consists of determining the number and type of serial operations required to complete the desired type of synchronization. Latencies for the various critical section algorithms,

Fetch&Φ implementations (coherent and remote word), and critical section states (idle and busy) are given in table 1.

Messages and Memory Operations for a Critical Section					
Algorithm		Idle		Busy	
		messages (n_{msg})	memory (n_{mem})	messages (n_{msg})	memory (n_{mem})
Anderson	coherent	12	12	9	10
	remote word	8	6	7	6.5
MCS	coherent	8	8	9	10
	remote word	8	6	5	7.5
QOLB	no prefetch	4	5	1	3
	prefetch	0	2	1	3
Messages and Memory Operations assuming Pairwise Sharing					
Algorithm		Idle		Busy	
		messages (n_{msg})	memory (n_{mem})	messages (n_{msg})	memory (n_{mem})
Anderson	coherent	6	9	5	7.5
	remote word	6	5	5	5.5
MCS	coherent	4	6	5	7.5
	remote word	6	5	3	6.5
Test&Set	coherent	4	6	5	8
	remote word	2	3	3	5.5
QOLB	no prefetch	2	4	1	3
	prefetch	0	2	1	3

Table 1: **Summary of Results.** The first half of the table summarizes the Anderson, MCS, and QOLB lock schemes in terms of critical section latency. Evaluations are made for both idle and busy critical sections as well as for both cache coherent and at-memory *Fetch&Φ* operations. The latency is subdivided into the numbers of interconnect messages and memory (including cache) accesses. The performance gain of using QOLB to prefetch a lock is also analyzed. In the second half of the table it is assumed that only two processes are sharing the lock and that they have entered SCI pairwise-sharing mode. For this situation, results for simple spin-locks (*Test&Set*) are also included.

Calculations in this table assume that $t_{msg} \gg t_{mem}$

For example, a write operation (or cache coherent *Fetch&Φ* operation) on a word in a line that is not local to the processor requires the following sequence of operations: (1) the initial miss in the requester's second level cache, (2) a request to the directory for the line, (3) an access to the directory, (4) a response yielding the current owner of the line, (5) a request to this node, (6) an access to fetch the line, (7) a reply containing the desired data, and (8) an access to place the line in the requester's cache. This results in four traversals of the interconnect and four memory accesses. The QOLB algorithm essentially requires these operations (plus an additional memory access for the initial QOLB operation) to acquire an idle critical section. Anderson's algorithm requires three such write operations when coherent *Fetch&Φ* operations are used to acquire an idle critical section.

Two significant improvements can be made to the lock algorithms. First, remote word primitives can be substituted for coherent line primitives. Although this analysis assumes that coherent line primitives

are always used to acquire the data, either type of primitive can be used for synchronization. If *Fetch&Φ* operations are performed at memory then only two traversals of the interconnect and a single memory access are required. Second, acquiring a line in pairwise sharing mode, eliminates the need to check the directory. Thus, only two traversals of the interconnect and three memory accesses are required. The memory accesses include the initial miss in the cache, the access to the other processor's cache, and the actual write into the requester's cache. It should be noted that pairwise sharing speeds up only coherent line operations.

The equation for calculating lock algorithm latency presented above requires two types of inputs. The first are the number of one-way interconnect traversals and memory accesses. These can be found in table 1. The second are latencies of typical traversals and memory accesses. These times must be estimated.

While memory access time can be treated as a constant for most purposes, interconnect traversal latency is dependent on a number of important factors, including interconnect architecture, size and loading. Thus, it is difficult to determine a precise value for t_{msg}. In the rest of our analysis we will only assume that it is greater than t_{mem}.

6.3 Analysis Results

Before comparing the various lock and data fetch algorithms, it is beneficial to choose among the two types of synchronization primitives (*i.e.* coherent and remote word). This is especially easy in the non-pairwise cases. From table 1 one sees that the remote word primitives clearly possess a performance advantage, because of the fewer number of interconnect traversals and memory accesses required to complete a *Fetch&Φ* operation. Only the MCS algorithm for idle locks gives even similar results. Thus, for the remainder of the study remote word operations will be used for these algorithms.

From the counts of traversals and memory accesses it is seen that QOLB has a significant performance advantage over the other lock algorithms. It is seen that QOLB (with no prefetch) is about twice as fast as the Anderson and MCS schemes for idle critical sections and four to fives times as fast for busy critical sections (*i.e.* lock transfer). If prefetching is possible, then QOLB's performance for idle locks improves by a much greater margin. The only latency involved is that for issuing the first QOLB, a second level cache access. These results are very similar to those from a simulation study of a different architecture [WG91].

Table 1 also shows results for situations involving pairwise sharing. In this case, the SCI-QOLB implementation produces a sizable increase in performance compared to all of the other algorithms. This benefit is, in all but one case, somewhat smaller for those algorithms that use remote word operations. For situations where the application knows in advance that a pairwise sharing situation exists, simple spin-locks using *Test&Set* and Unset may be used. With only two processes actively contending for a lock, one of which will typically already possess the lock, *Test&Set* is quite efficient. In fact, *Test&Set* will outperform the other software queuing methods due to its low overhead. *Test&Set* and QOLB are approximately equal when a critical section is idle and prefetching is not used.

QOLB's largest performance advantage is realized when there is no contention for the lock. In this case QOLB can prefetch and have data ready in the cache when the critical region is entered. This is shown in table 1 as the two cases where there is a zero message count. In this case the MSC algorithm uses between 4 and 8 interconnect traversals, depending upon other parameters.

To sum up this section we conclude that QOLB hardware in the SCI protocol decreases the number of interconnect traversals in order to acquire a lock and the related shared date by a large factor. When there is contention for the data, QOLB uses only one interconnect traversal to fetch both the lock and the data. In the absence of contention the number of traversals is zero if prefetch is possible and two (pairwise sharing) or four (no pairwise sharing) when prefetch is not possible. These low number of traversals will give the programmer of a shared memory SCI-based computer the possibility to get useful work out of a lot of processors.

7 Summary

We have discussed and analyzed an implementation of the QOLB primitives based on the Scalable Coherent Interface. Since both QOLB and the base SCI cache coherence protocol are based on a linked-list directory, the starting point for implementing QOLB in SCI is very good. An SCI-based system can be implemented with some nodes supporting the QOLB option while others do not.

We have addressed a number of SCI-specific issues. Particular attention was paid in the implementation to ensure that knowledge of QOLB is not needed by processes that do not wish to use that primitive. In addition the definition of QOLB allows its implementation as a no-op.

The use of pointers linked through caches to form QOLB queues allows cache lines to be rolled out, and our implementation is made much more complex by the possibility of a queue breaking down. For this reason, a state indicating QOLB history information is added to ensure that the effect of cyclic patterns causing excess spinning is minimized. Use of the QOLB primitives in a program running on an SCI-based multiprocessor is also made more complex because a line can disappear from the cache at any time. Such complexity is mainly handled once and for all, and then hidden in library routines, some of which have been given in this paper. Additional concerns such as process migration and forward progress have also been addressed.

The authors are convinced that hardware support for process communication in a shared memory multiprocessor is necessary for fined grained program execution to be efficient on such computers. As a first step to investigate this hypothesis we have in this paper analyzed two of the most important concurrent programming scenarios.

We have compared the SCI-QOLB implementation to two software schemes [And90, MCS91]. The performance of the software schemes is dependent on the coherence protocol of the hardware used, and for our comparison we have assumed they use the base SCI cache coherence protocol. QOLB reduces latency over these schemes by a factor of two to four, although much greater improvements are possible if QOLB is used both to synchronize on and prefetch the lock and shared data associated with a critical section.

By using the SCI cache-to-cache write transaction that moves a memory line copy from one cache to another, as well as the pairwise sharing optimization, the QOLB mechanism for the exchange of critical section ownership reduces latency to a small number of interconnect traversals and memory accesses. By using the doubly-linked SCI sharing list to queue waiters on a critical section, we also allow an enqueued processor to change its mind and leave the waiting queue before acquiring the lock.

We have also analyzed a producer/consumer situation. We demonstrated how QOLB can be used to prefetch shared data, completely eliminating interconnect latency. We have shown that such a situation exists when a producer has released the data before a consumer starts using it and the consumer has an outstanding prefetch on this data. In this case the consumer acquires the data without any network delay – it simply accesses its cache and finds that the data is already there. Such overlap of execution and data transport will typically happen when it is needed the most, that is, for critical path processes. When all predecessors in the dependency graph are completed, the critical path process (having issued a prefetched on the data) can immediately continue doing useful work, assuming the needed data is released before the last predecessor completes. If the program is such that processes along the critical path can be split up into yet smaller processes that communicate and prefetch using QOLB, a finer execution granularity may be achieved that could yield even higher performance.

The results presented in this paper agree well with those of an earlier study [WG91] in which the performance benefits of QOLB were quantified for a bus-based architecture. The results demonstrate that there is much to be gained by providing hardware support for critical sections. While simulation studies of real programs might demonstrate more clearly the advantages to be gained by the use of QOLB, they still would be unable to assess the real value of QOLB since the algorithms were not designed to exploit such hardware support.

To demonstrate our theses, a model of a full SCI system and a C-compiler that can handle the QOLB primitives are under development. The compiler and SCI system are nearing completion. The next task will be to re-write current applications and analyze their new performance.

In addition, QOLB provides opportunities for additional relaxation of the consistency model, which suggests further advantages. These issues are the subject of continuing study.

8 Acknowledgements

This work is supported by the National Science Foundation grant #CCR-892766 and by the Norwegian research council NTNF. We wish to acknowledge the enormous contributions of David James, to the Scalable Coherent Interface in general, and to the QOLB aspects specifically. David Gustavson, Ross Johnson, Stein Krogdahl and Ernst Kristiansen, along with numerous other members of the SCI Working Group, have contributed much to the development and understanding of the concepts developed in this study.

References

[AB86] J. Archibald and J.-L. Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". *Transactions on Computer Systems*, pages 273–298, November 1986.

[And90] T. E. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". *IEEE Transactions on Parallel and Distributed Systems*, pages 6–16, January 1990.

[GLL+90] K. Gharacharloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors". In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 63–79. ACM, May 1990.

[GVW89] J. R. Goodman, M. K. Vernon, , and P. J. Woest. "A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor". In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*. ACM, April 1989.

[IEE91] IEEE Microprocessors Standards Committee. "Scalable Coherent Interface: Logical, Physical and Cache Coherence Specifications". P1596 Working Group of the Microprocessors Standards Committee, November 1991.

[JLG+90] D. V. James, A. T. Laundrie, S. Gjessing, , and G. S. Sohi. "Scalable Coherent Interface". *IEEE Computer*, 23(6):74–77, June 1990.

[Jor83] H. F. Jordan. "Performance Measurements on HEP – a Pipelined MIMD Computer". In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 207–212. ACM, June 1983.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 269–278. ACM, April 1991.

[MG91] T. Mowry and A. Gupta. "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors". *Journal of Parallel and Distributed Computing*, pages 87–106, June 1991.

[RS84] L. Rudolph and Z. Segall. "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors". In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347. ACM, June 1984.

[Sto90] H. S. Stone. *"High-Performance Computer Architecture"*. Addison-Wesley, 2nd edition, 1990. Page 362.

[WG91] P. J. Woest and J. R. Goodman. "An Analysis of QOLB Synchronization and Prefetching in Shared-Memory Multiprocessors". In *Proceedings of the International Symposium on Shared-Memory Multiprocessors*, April 1991. Also as Technical Report 1005, Computer Sciences Department, University of Wisconsin-Madison, February 1991.

[YTL87] P. C. Yew, N. F. Tzeng, and D. H. Lawrie. "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors". *IEEE Transactions on Computers*, pages 388–395, April 1987.